Compiling with Generating Functions

JIANLIN LI, University of Waterloo, Canada
YIZHOU ZHANG, University of Waterloo, Canada

We present a new approach to scaling exact inference for probabilistic programs, using generating functions (GFs) as a compilation target. Existing methods that target representations like binary decision diagrams (BDDs) achieve strong state-of-the-art results. We show that a compiler targeting GFs can be similarly competitive—and, in some cases, more scalable—on a range of inference problems where BDD-based methods perform well.

We present a formal model of this compiler, providing the first definition of GF compilation for a functional probabilistic language. We prove that this compiler is correct with respect to a denotational semantics. Our approach is implemented in a probabilistic programming system called Geni and evaluated on a range of inference problems. Our results establish GF compilation as a principled and powerful paradigm for exact inference: it offers strong scalability, good expressiveness, and a solid theoretical foundation.

CCS Concepts: • **Mathematics of computing → Generating functions**; *Probabilistic representations*; **Probabilistic reasoning algorithms**; *Statistical software*; *Bayesian computation*; *Mathematical software performance*; • **Theory of computation** → *Probabilistic computation*; *Denotational semantics*; • **Computing methodologies** → *Probabilistic reasoning*; • **Software and its engineering → Compilers**; *Functional languages*; *Domain specific languages*; *Formal language definitions*; *Semantics.*

Additional Key Words and Phrases: Probabilistic programming languages, probabilistic inference, probabilistic verification, compiler correctness.

ACM Reference Format:
Jianlin Li and Yizhou Zhang. 2025. Compiling with Generating Functions. *Proc. ACM Program. Lang.* 9, ICFP, Article 265 (August 2025), 25 pages. https://doi.org/10.1145/3747534

1 Introduction

Systems with probabilistic behavior are ubiquitous: from computer networks to biological processes to economic models. Probabilistic programming languages (PPLs) provide a high-level abstraction for modeling such systems. Given a probabilistic program, the PPL performs *probabilistic inference* to answer queries about the program's behavior—typically, the probability that an event occurs according to the distribution defined by the program.

Probabilistic inference is computationally challenging, however. Exact inference is even more so, as it is concerned with computing precise probabilities rather than approximations. This paper focuses on the computational challenge of exact inference for discrete probabilistic programs. Even for a subclass of discrete programs where all variables have finite support, exact inference is known to be PSPACE-hard in the worst case.

Authors' Contact Information: Jianlin Li, David R. Cheriton School of Computer Science, University of Waterloo, Ontario, Canada, jianlin.li@uwaterloo.ca; Yizhou Zhang, David R. Cheriton School of Computer Science, University of Waterloo, Ontario, Canada, yizhou@uwaterloo.ca.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2475-1421/2025/8-ART265
https://doi.org/10.1145/3747534

Compact representations. Advances have been made in exact inference for discrete probabilistic programs. The ability to represent probability distributions in a compact form is perhaps the most important factor in scaling exact inference—the vanilla approach of using conditional probability tables (CPTs) quickly becomes intractable as the number of variables increases.

The Dice PPL [19] represents one approach to scalable exact inference. It compiles probabilistic programs to *binary decision diagrams* (BDDs) [4], which can succinctly represent the distribution defined by the program. Dice has been applied to probabilistic model checking [18], where it is shown to outperform state-of-the-art model checkers such as Storm [17] on some benchmarks.

Another direction that has been explored recently [21, 8, 36, 22] is the use of *generating functions* (GFs) [35] for exact inference. Generating functions are a powerful mathematical tool that can succinctly represent probability distributions over natural numbers. They also enable exact inference in the presence of infinite-support distributions (e.g., Poisson), which is not possible with BDDs.

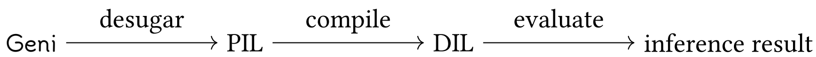
Although GFs are mathematically appealing, their use in practical PPL implementations faces challenges. First, current systems are concerned with *imperative* probabilistic programs; it is not immediately clear how to systematically compile *functional* probabilistic programs to GFs and further to state and prove correctness of such a compilation. Second, we observe that existing GF-based PPL implementations do not scale well for many programs that Dice can handle effectively. For example, Genfer [36] is a GF-based PPL that has been shown to outperform some other GF-based implementations on a range of benchmarks, but it struggles to scale beyond programs that contain more than a few variables.

BDD-based inference, in contrast, is effective for more complex programs and, thus, stands as a common, established approach to exact inference. At first glance, it may seem unlikely that GFs could compete with BDDs as a compilation target for scaling inference for probabilistic programs over binary random variables. After all, generating functions are typically valued for their ability to encode and manipulate possibly infinite sequences.

Contributions. This paper presents a compiler using generating functions as the compilation target and offering strong scalability, good expressiveness, and a solid theoretical foundation.

Several factors contribute to the scalability of our approach. Generating functions enjoy properties that allow a compiler to effectively exploit independence and determinism in program structure. Reducing derivative computations, as well as applying simple compiler and runtime optimizations, also improves inference scalability.

We implement our approach in a probabilistic programming system called Geni. Geni is a functional PPL supporting finite- and infinite-support discrete distributions, continuous priors, Bayesian observations, bounded loops, and a restricted form of unbounded loops. The implementation of Geni is structured as follows:



The central component of the Geni implementation is the compilation from PIL to DIL.

- Section 4 formalizes the syntax and semantics of PIL, a functional probabilistic intermediate language. High-level constructs in the Geni surface language, such as logical operations, integer arithmetic, bounded loops, and non-recursive functions, are desugared to PIL. In PIL, all values are either natural numbers or tuples of natural numbers, so PIL is amenable to compilation to generating functions.
- Section 5 formalizes the compilation of PIL programs into a deterministic language, called DIL, that we use to specify generating functions. We establish the correctness of this compilation

into GFs. This formalization is novel, as it is the first time that a GF compilation is defined for a *functional* PPL and proven correct. The close correspondence between the PIL denotational semantics and the PIL-to-DIL compilation is a key insight that helps clarify correctness.

- Section 6 describes the implementation of Geni.
- Section 7 evaluates Geni on a range of benchmarks. When compared to state-of-the-art results from Dice, Geni demonstrates comparable performance—and sometimes, better scalability on some inference problems where Dice has been considered effective—while also allowing exact solutions to be derived automatically for a wider range of programs.

We proceed by first reviewing generating functions in the context of probability theory (Section 2) and then illustrating the key ideas in the design and implementation of Geni with examples (Section 3).

2 Background on Generating Functions

A brief review of measure theory relevant to our development can be found in an appendix in the extended version [28]. In this section, we review the concept of generating functions.

Probability-generating functions are a common tool for studying random variables and their distributions, particularly discrete ones. For a random variable X supported on $\mathbb{N}$ or a subset thereof, its probability-generating function $\mathbb{G}_X : \mathbb{R} \to \mathbb{R}$ is defined as

$$\mathbb{G}_X(x) = \mathbb{E}_X\left[x^X\right] = \sum_{n\in\mathbb{N}} \mathbb{P}[X = n] \cdot x^n. \tag{2.1}$$

Here, the indeterminate x is the formal parameter of the probability-generating function of X. We use uppercase letters for random variables and lowercase letters for their corresponding GFs' formal parameters. For instance, the GF of a Bernoulli random variable X with parameter 0.1 is

$$\mathbb{G}_{X\sim\text{Bernoulli}(0.1)}(x) = 0.9 + 0.1x. \tag{2.2}$$

In Bayesian probabilistic programming, we often work with unnormalized measures. For a discrete measure μ supported on $\mathbb{N}$ or a subset of $\mathbb{N}$, we call $\mathbb{G}_X(x) = \mathbb{E}_{X\sim\mu}\left[x^X\right]$ the *generating function* (GF) of μ, omitting the term "probability" since μ may not be a proper probability measure. We use the notation $\mathbb{G}_\mu$ or $\mathbb{G}_X$ to mean the GF of μ or X.

Probability masses can be recovered from the GF by taking derivatives:

$$\mathbb{P}[X = n] = \frac{1}{n!} \cdot \mathbb{G}_X^{(n)}(0), \tag{2.3}$$

which explains the term "probability-generating function". Here, $\mathbb{G}_X^{(n)}(x)$ denotes the n-th derivative of $\mathbb{G}_X$ at point x. The power series $\sum_{n\in\mathbb{N}}(1/n!) \cdot \mathbb{G}_X^{(n)}(0) \cdot x^n$ is precisely the Taylor expansion of $\mathbb{G}_X$ at 0.

Moments can also be obtained from GFs by taking derivatives. In particular, the expectation (i.e., the first raw moment) and the variance (i.e., the second central moment) of a random variable X can be obtained as follows:[1]

$$\mathbb{E}[X] = \mathbb{G}_X^{(1)}(1) \qquad \text{and} \qquad \mathbb{V}[X] = \mathbb{G}_X^{(2)}(1) + \mathbb{G}_X^{(1)}(1) - \mathbb{G}_X^{(1)}(1)^2. \tag{2.4}$$

For the Bernoulli example, one can verify that the probabilities and expected value check out:

$$\mathbb{P}[X = 0] = \tfrac{1}{0!} \cdot \mathbb{G}_X^{(0)}(0) = 0.9 \qquad \mathbb{P}[X = 1] = \tfrac{1}{1!} \cdot \mathbb{G}_X^{(1)}(0) = 0.1 \qquad \mathbb{E}[X] = \mathbb{G}_X^{(1)}(1) = 0.1$$

[1]Moments may not exist for some distributions. For example, the generalized Zipf distribution with exponent 2 has the generating function $\mathbb{G}_{X\sim\text{Zipf}(2)}(x) = \zeta(2)^{-1} \cdot \text{Li}_2(x)$, where $\text{Li}_2(x)$ is the polylogarithm function of order 2. The derivative $\mathbb{G}^{(1)}_{X\sim\text{Zipf}(2)}(1)$ diverges—the mean of Zipf(2) does not exist. A sufficient condition for the existence of moments is exponential tail decay, which is satisfied by the distributions we consider.

The GF of a multidimensional random variable $\boldsymbol{X} = (X_1, \dots, X_d)$ is defined as a multivariate GF:

$$\mathbb{G}_{\boldsymbol{X}}(\boldsymbol{x}) = \mathbb{G}_{\boldsymbol{X}}(x_1, \dots, x_d) = \mathbb{E}_{\boldsymbol{X}}\left[\prod_{i=1}^{d} x_i^{X_i}\right]. \tag{2.5}$$

We often write $\boldsymbol{x}^{\boldsymbol{X}}$ as shorthand for $\prod_{i=1}^{d} x_i^{X_i}$. Setting x_i to 1 in the multivariate GF has the effect of marginalizing out the i-th dimension, yielding the GF of the remaining dimensions:

$$\mathbb{G}_{\boldsymbol{X}\setminus X_i}(x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_d) = \mathbb{G}_{\boldsymbol{X}}(x_1, \dots, x_{i-1}, 1, x_{i+1}, \dots, x_d). \tag{2.6}$$

For a random variable Y obtained by an affine transformation of X (i.e., $Y = m \cdot X + n$, where $m, n \in \mathbb{N}$), the GF of the joint distribution of X and Y is

$$\mathbb{G}_{XY}(x, y) = y^n \cdot \mathbb{G}_X(x \cdot y^m). \tag{2.7}$$

For two independent random variables X and Y, the GF of their sum $Z = X + Y$ is the product of their GFs at point z:

$$\mathbb{G}_Z(z) = \mathbb{G}_X(z) \cdot \mathbb{G}_Y(z). \tag{2.8}$$

For independent X and Y, the GF of their joint distribution is the product of the GFs:

$$\mathbb{G}_{XY}(x, y) = \mathbb{G}_X(x) \cdot \mathbb{G}_Y(y). \tag{2.9}$$

In our development, we work with a language supporting pairs, so we generalize the notion of GFs by allowing pairs (as opposed to real vectors) as inputs. For example, consider the program $\varnothing \vdash \mathsf{let}\ X = \mathsf{flip}\ 0.1\ \mathsf{in}\ (X, 8 \cdot X + 1) : \mathbb{N} \times \mathbb{N}$. This program returns a pair: it returns $(1, 9)$ with probability 0.1 and $(0, 1)$ with probability 0.9. If we let Y be the random variable representing the return value of this program, then we consider its GF $\mathbb{G}_Y$ to take a pair of real numbers as input:

$$\mathbb{G}_Y(y) = \textbf{let}\ y_1 = \textbf{fst}\ y\ \textbf{in}\ \textbf{let}\ y_2 = \textbf{snd}\ y\ \textbf{in}\ 0.1 \cdot y_1 \cdot y_2^9 + 0.9 \cdot y_2.$$

In our development, we will use GFs to represent measures over semantic substitutions for free variables in programs. So we will further generalize the notion of GFs to allow substitutions as inputs (Section 5.2).

Common infinite-support discrete distributions have closed-form GFs. For example, the GFs of geometric distributions and Poisson distributions are given by

$$\mathbb{G}_{X\sim\mathsf{Geo}(\theta)}(x) = \theta/(1 - (1-\theta)x) \quad \text{and} \quad \mathbb{G}_{X\sim\mathsf{Poisson}(\theta)}(x) = \exp(\theta(x-1)). \tag{2.10}$$

GFs are best known for their ability to compactly encode, manipulate, and analyze infinite sequences.

Common continuous distributions have GFs too. For a random variable X supported on $\mathbb{R}$ or a subset thereof, its GF $\mathbb{G}_X : \mathbb{R} \to \mathbb{R}$ is defined as

$$\mathbb{G}_X(x) = \mathbb{E}_X\left[x^X\right] = \int_{\mathbb{R}} x^X \, \mathrm{d}X. \tag{2.11}$$

For example, an exponential distribution $\mathsf{Exp}(\theta)$ has GF $\mathbb{G}_{X\sim\mathsf{Exp}(\theta)}(x) = \theta/(\theta - \log x)$, and its mean can be computed as $\mathbb{E}_{X\sim\mathsf{Exp}(\theta)}[X] = \mathbb{G}^{(1)}_{X\sim\mathsf{Exp}(\theta)}(1) = 1/\theta$.

3 Main Ideas

Consider the PIL program in the left column of Figure 1. The construct $\mathsf{flip}\ \theta$ samples a Bernoulli random variable supported on $\{0, 1\}$ with parameter θ. The program returns 1 if the sum S_n of the n variables $X_1, \dots, X_n$ is positive and 0 otherwise. In other words, the program encodes the disjunction of n Bernoulli random variables: $X_1 \vee X_2 \vee \dots \vee X_n$. Although the program is simple and artificial, it serves as a good starting point to illustrate the main ideas. We will evaluate Geni on more complex programs further on.

PIL program	**DIL expressions**	**Generating functions**
	$e_0 := 1$	1
let X_1 = flip θ_1 in	$e_1 := e_0 \cdot \left(\overline{\theta_1} + \theta_1 \cdot x_1\right)$	$\lambda x_1 . \overline{\theta_1} + \theta_1 x_1$
let $S_1 = X_1$	$e_1' := e_1[x_1 \mapsto x_1 \cdot s_1][x_1 \mapsto 1]$	$\lambda s_1 . \overline{\theta_1} + \theta_1 s_1$
let X_2 = flip θ_2 in	$e_2 := e_1' \cdot \left(\overline{\theta_2} + \theta_2 \cdot x_2\right)$	$\lambda s_1 x_2 . \left(\overline{\theta_1} + \theta_1 s_1\right)\left(\overline{\theta_2} + \theta_2 x_2\right)$
let $S_2 = S_1 + X_2$	$e_2' := e_2[s_1 \mapsto s_1 \cdot s_2, x_2 \mapsto x_2 \cdot s_2]$ $[s_1 \mapsto 1][x_2 \mapsto 1]$	$\lambda s_2 . \left(\overline{\theta_1} + \theta_1 s_2\right)\left(\overline{\theta_2} + \theta_2 s_2\right)$
let X_3 = flip θ_3 in	$e_3 := e_2' \cdot \left(\overline{\theta_3} + \theta_3 \cdot x_3\right)$	$\lambda s_2 x_3 . \left(\overline{\theta_1} + \theta_1 s_2\right)\left(\overline{\theta_2} + \theta_2 s_2\right)\left(\overline{\theta_3} + \theta_3 x_3\right)$
let $S_3 = S_2 + X_3$	$e_3' := e_3[s_2 \mapsto s_2 \cdot s_3, x_3 \mapsto x_3 \cdot s_3]$ $[s_2 \mapsto 1][x_3 \mapsto 1]$	$\lambda s_3 . \left(\overline{\theta_1} + \theta_1 s_3\right)\left(\overline{\theta_2} + \theta_2 s_3\right)\left(\overline{\theta_3} + \theta_3 s_3\right)$
...	...	...
	$e_\perp := e_n'[s_n \mapsto 0][s_n \mapsto 1]$	$\overline{\theta_1} \cdots \overline{\theta_n}$
if S_n then	$e_\top := e_n'[s_n \mapsto 1] - e_\perp$	$1 - \overline{\theta_1} \cdots \overline{\theta_n}$
1	$e_\top' := e_\top \cdot z$	$\lambda z . \left(1 - \overline{\theta_1} \cdots \overline{\theta_n}\right) z$
else		
0	$e_\perp' := e_\perp \cdot z^0$	$\lambda z . \overline{\theta_1} \cdots \overline{\theta_n}$
	$e' := e_\top' + e_\perp'$	$\lambda z . \left(1 - \overline{\theta_1} \cdots \overline{\theta_n}\right) z + \overline{\theta_1} \cdots \overline{\theta_n}$

Figure 1. **Left**: A program written in PIL syntax. **Center**: The compiled program in DIL. Each line is a DIL expression that represents the GF in the variables still live after the line in the PIL program. The substitutions $[x \mapsto 1]$ in gray have the effect of marginalizing out a variable X whose liveness is ending. **Right**: Denotational semantics of the DIL expressions in the center.

GF compilation. Let Z be the random variable denoting the return value of the PIL program. The right column of Figure 1 shows the step-by-step derivation of the GF of Z. The final GF is

$$\mathbb{G}_Z(z) = \left(1 - \overline{\theta_1} \cdots \overline{\theta_n}\right) z + \overline{\theta_1} \cdots \overline{\theta_n}, \text{ where } \overline{\theta_i} = 1 - \theta_i, \tag{3.1}$$

which encodes the marginal distribution of Z. The goal is to *automate* the derivation of a computable form of the generating function from a given PIL program.

To achieve this goal, we need to define a translation from PIL to a target language that can express generating functions. We call this target language DIL. While PIL is a probabilistic language, DIL is deterministic. The result of this compilation is the expression e' defined in the center column of Figure 1. Each line in the center column is a subexpression of e' constructed during the compilation process.

If the compilation is correct, the final DIL expression e' should have a denotational semantics that is precisely the generating function $\mathbb{G}_Z$. It is in this sense that we refer to the PIL-to-DIL compilation process as *GF compilation*.

The curious reader may want to take a sneak peek at the commutative diagram in Figure 8, which illustrates the compiler correctness property (Theorem 5.1) we will establish.

GF compilation for a functional language. Existing PPLs that compile to GFs are imperative [36, 22]: the programs are sequences of commands that mutate a global store of a fixed set of k variables. This design is convenient, as it allows the use of textbook definition of GFs, which operate on fixed-size vectors. All GFs constructed during translation have the same k-dimensional vector space as their domain, and each command acts as a transformer between two k-variate GFs—mirroring Kozen's seminal distribution-transformer semantics for imperative probabilistic programs [23].

We work in a functional setting, which improves modularity but complicates GF compilation. In our functional setting, the set of in-scope variables is not fixed, so the type signatures of GFs must vary with the typing context. Moreover, data types in PIL include nested tuples, which prior work on GF compilation does not consider.

To support all this flexibility, we generalize the notion of GFs, so that they operate on semantic substitutions for typing contexts rather than fixed-size vectors (Section 5.2).

We can then view GF compilation as a symbolic execution of the functional PIL program, where the symbolic representations denote generalized GFs. That is, the DIL expressions constructed during compilation are symbolic GFs representing measures over the possible *program states*—valuations of the in-scope variables—at each program point. So we can view these PIL expressions as transformers between symbolic representations of in-scope variables, with the ability to introduce and eliminate variables as scopes change.

We use this GF-transformer view to define the GF compilation for PIL (Section 5), and the measure-transformer view to define its denotational semantics (Section 4), connecting them via the generalized notion of GFs. The payoff is a tight correspondence between the denotational semantics and the GF compilation, which elucidates essence and justifies correctness.

Marginalizing out expired variables. A key optimization in the Geni compiler is the marginalization of variables when they expire—i.e., when they are no longer live. This optimization reduces the input sizes of the intermediate GF expressions created during compilation, which can lead to savings in time and space during inference.

As shown in (2.6), marginalizing out a variable X_i from a multivariate GF $\mathbb{G}_{\boldsymbol{X}}$ is as simple as setting x_i to 1. In the DIL program in Figure 1, this marginalization is performed by the substitutions $[x_i \mapsto 1]$ and $[s_i \mapsto 1]$ for $i = 1, 2, \ldots, n$.

Importantly, these substitutions occur in the compiled DIL program *as soon as* the variable's liveness ends in the PIL program. If the marginalization were delayed until the end of the program, the following DIL expression would be generated:

$$\left(\overline{\theta_1} + \theta_1 x_1 s_1 s_2 \cdots s_n\right)\left(\overline{\theta_2} + \theta_2 x_2 s_2 \cdots s_n\right) \cdots \left(\overline{\theta_n} + \theta_n x_n s_n\right)[x_1 \mapsto 1][s_1 \mapsto 1] \cdots [x_n \mapsto 1][s_n \mapsto 1] \quad (3.2)$$

Evaluating this expression requires $O(n^2)$ time, as the k-th last factor in the product involves $O(k)$ operations. By contrast, the expression e' defined in Figure 1 can be evaluated more efficiently in $O(n)$ time.

Extracting probabilities from GFs without differentiation. The Genfer PPL [36] computes partial derivatives of generating functions, whenever compilation encounters a conditional statement (if E then ... else ...) or Bayesian conditioning statement (observe E), to extract the probability that the conditional expression E evaluates to true. This differentiation is identified as a performance bottleneck of Genfer.

Geni reduces the need for differentiation: derivatives are not required when compiling conditionals that test if a variable is nonzero. We make the observation of a perhaps obvious fact: for a random variable X supported on $\mathbb{N}$ or a subset, the probability masses of the zero and non-zero

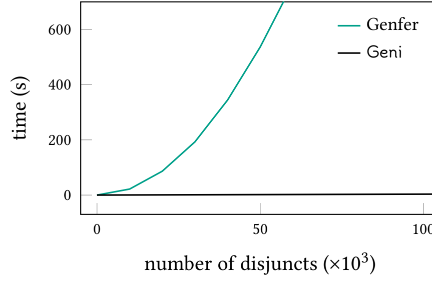


Figure 2. Scaling of Genfer and Geni on the program from Figure 1.

events can be directly read off from its generating function $\mathbb{G}_X(x)$ without differentiation. That is,

$$\mathbb{P}[X = 0] = \frac{1}{0!} \cdot \mathbb{G}_X^{(0)}(0) = \mathbb{G}_X(0) \quad \text{and} \quad \mathbb{P}[X \neq 0] = 1 - \mathbb{P}[X = 0] = 1 - \mathbb{G}_X(0). \tag{3.3}$$

More generally, for a measure μ supported on $\mathbb{N}$ or a subset, we have

$$\mu(\{0\}) = \mathbb{G}_\mu(0) \quad \text{and} \quad \mu(\mathbb{N} \setminus \{0\}) = \mathbb{G}_\mu(1) - \mathbb{G}_\mu(0). \tag{3.4}$$

The compilation of the if expression in Figure 1 uses this fact to extract the masses ($e_\top$ and $e_\bot$) without differentiation. All that is needed is to evaluate the GF at 0 and 1 and perform subtraction.

This observation allows Geni to avoid differentiation for a large class of programs—all programs that can be expressed by Dice. It is easy to see that differentiation is not needed for a conditional expression testing if a variable is nonzero. For example, consider the compilation of the if expression in Figure 1. There, the GF expression e'_n is the symbolic representation of program execution before the if statement. Geni transforms e'_n into two GF expressions $e_\top$ and $e_\bot$, which constrain e'_n to the cases where S_n is true and false, respectively. The else branch GF $e_\bot$ is obtained by setting s_n to 0 in e'_n, while the then branch GF $e_\top$ is obtained by GF subtraction—differentiation is not required.

For finite-support integer variables, Geni supports them in the same way as Dice: by encoding them as tuples of binary variables [5]. So differentiation is not needed for conditional expressions testing such variables. A variable X supported on, say, $\{0, 1, 2, 3\}$ is encoded as a pair of binary variables (X_1, X_2), where $X = 2X_2 + X_1$. Testing $X = 2$ does not require computing the second-order derivative $(1/2!) \cdot \mathbb{G}_X^{(2)}(0)$. Instead, it can be done by testing $X_1 = 0 \wedge X_2 = 1$ without evaluating any derivatives:

$$\mu_{X_1X_2}\big(\{(0,1)\}\big) = \mu_{X_1}\big(\{0\}\big) - \mu_{X_1X_2}\big(\{(0,0)\}\big) = \mathbb{G}_{\mu_{X_1X_2}}(0,1) - \mathbb{G}_{\mu_{X_1X_2}}(0,0). \tag{3.5}$$

Here, $\mu_{X_1X_2}$ is the joint measure of (X_1, X_2) and μ_{X_1} is the marginal measure of X_1.

Figure 2 compares how Geni and Genfer scale on the program from Figure 1. Geni scales linearly with the size of the program, whereas Genfer scales super-linearly.[2] This difference arises because Genfer evaluates the n-th derivative of the GF, which requires time polynomial in n, as explained by Zaiser et al. [36], whereas Geni avoids differentiation entirely.

[2]In this experiment, Genfer is run on a manually modified version of the program from Figure 1, which we obtained by manually performing a live variable analysis. This modified version recycles variables when their liveness ends. The result is a program that uses only two variables, X and S, updated imperatively throughout the program. Using a constant number of variables is favorable to Genfer's performance; without it, Genfer would scale exponentially. Nevertheless, we mention that there exist encodings of logical disjunction on which Genfer scales linearly.

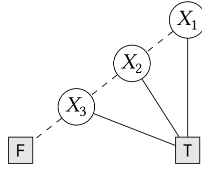


Figure 3. The BDD that the Dice compiler [19] generates for the disjunction $X_1 \vee X_2 \vee X_3$. Dashed lines indicate that the parent variable takes the value F (false), while solid lines indicate T (true). Dice's BDD compilation exploits the determinism in this model—the size of the BDD increases linearly with the number of variables.

Scaling exact inference. Exact inference for discrete models, even without infinite-support distributions, is PSPACE-hard. Dice [19] is the state of the art in scaling exact inference for such probabilistic programs. How does Geni compare to Dice on this front? In Section 7, we seek to answer this question based on empirical evidence, by evaluating Geni on benchmarks that Dice is known to handle effectively. The results indicate that Geni can exploit these structures reasonably well, too—and, perhaps surprisingly, even better than Dice in some cases. Here, we preview the findings.

Previous work [7] suggests that scaling exact inference for discrete models requires exploiting available *independence structure* and *local structure* in the models.

Independence structure refers to conditional independence relations among variables, like the kind available in Markov models. It allows computations in inference to be factorized into smaller computations. Generating functions automatically encode these relations to some extent. As (2.8) and (2.9) show, independence relations automatically lead to factorized GFs. In addition, by marginalizing out variables as soon as they are no longer dependent upon, Geni keeps the sizes of the GFs as small as possible. Other exact-inference methods, including BDD compilation and those based on variable elimination [37, 16, 27], are known to exploit this structure well.

Local structure refers to a broad category of structural properties including, most notably, determinism. Determinism often exists in probabilistic models with explicit logical conditions, such as those found in probabilistic logic programs and computer network models. Dice [19] exploits determinism well: determinism leads to sparse CPTs, which in turn lead to compact BDDs. For example, the disjunction of n binary variables $X_1 \vee \ldots \vee X_n$ is a model with a high degree of determinism—it is used as an example by Holtzen et al. [19] to illustrate local structure. As shown in Figure 3, Dice can compile this model to a BDD of size $O(n)$, rather than a complete binary tree of size $O(2^n)$. By contrast, it is well understood that methods based on variable elimination fail to exploit this determinism, solving this inference problem in $O(2^n)$ time.

Geni can effectively exploit this determinism, too, with the encoding shown in Figure 1. In fact, as shown in Figure 4, Geni scales better than Dice on this example. Geni scales linearly with the number of disjuncts n, while Dice scales super-linearly.[3] Interestingly, our empirical finding seems at odds with the comment by Holtzen et al. [19] on the disjunction model: "if the number of variables disjoined together were to increase, the size of the BDD—and the cost of compiling it—would increase only linearly with the number of variables." We believe the nuance is that while the BDD size increases linearly in this case, constructing the BDD may involve operations that require super-linear time in general [4].

Nevertheless, while Geni is effective at exploiting determinism, it may not be as effective as Dice at handling programs translated from *conditional probability tables*. Handling large CPTs well

[3]We also experimented with alternative encodings of disjunction in Dice. Dice scales super-linearly on these encodings, while Geni scales linearly.

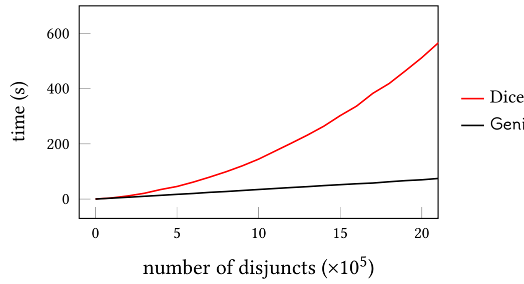


Figure 4. Scaling of Dice and Geni on disjunction encodings. Geni is run on the same program from Figure 1. The same program cannot be expressed in Dice, so we run Dice on a similar program with all summations replaced by disjunctions.

requires extracting local structure implicitly present in the tabular representation, which BDD compilation largely automates [19]. As a result, Dice scales better than Geni on such benchmarks. Despite this limitation, we observe that Geni can still compute exact solutions for very large Bayesian networks specified as CPTs in reasonable time, where Genfer and methods like variable elimination would time out or run out of memory in practice.

In summary, our empirical results in Section 7 indicate that Geni and Dice have complementary strengths. On some inference problems that motivated the use of Dice [18, 33], Geni may scale surprisingly well: where Dice scales super-linearly, Geni may scale linearly; and where both scale exponentially, Geni may do so with a smaller base. These models often feature a good amount of independence and determinism. But Dice may outperform Geni on some other inference problems, particularly those with large, dense CPTs. We view these findings as a first step toward understanding the performance characteristics of GF-based inference versus BDD-based inference; a more definitive picture will require further investigation.

Handling infinite-support distributions. Infinite-support distributions naturally arise in probabilistic models in domains like evolutionary biology and network traffic analysis. For example, consider the program below written in Geni:

$$\begin{array}{l}\mathsf{let}\ X = \mathsf{sample\ Poisson}(1)\ \mathsf{in}\\ \mathsf{loop}\ X\ \mathsf{sum}\ \mathit{ForwardPacket}()\end{array}$$

This program models network traffic where packets arrive according to a Poisson distribution, which is a standard assumption in traffic modeling. The packets are forwarded through a network of routers. The function *ForwardPacket*() may be arbitrarily complex: it models the network topology and also the probabilistic behavior of the router network—routers and links may fail probabilistically. *ForwardPacket* returns 1 if a packet reaches the destination and 0 otherwise. The expression loop X sum *ForwardPacket*() simulates forwarding all X packets through the network and computes the total number of packets that are successfully transmitted.

Programs like this are challenging for current PPLs. Dice cannot handle the infinite-support Poisson distribution. Genfer supports Poisson distributions, but it struggles to scale to complex network models and does not support loops. Sampling-based methods, such as importance sampling and Hamiltonian Monte Carlo in PPLs like Pyro [2] and Stan [6], struggle with this inference problem due to the high-dimensional discrete structure and inherent non-differentiability.

$$\begin{array}{rrcl}
\text{type} & T & ::= & \mathbb{N} \mid T \times T \\
\text{expression} & E & ::= & X \mid n + \sum_i n_i \cdot X_i \mid (E_1, E_2) \mid \mathsf{fst}\ E \mid \mathsf{snd}\ E \mid \mathsf{let}\ X = E_1\ \mathsf{in}\ E_2 \mid X \in A \mid \\
& & & \mathsf{sample}\ D \mid \mathsf{observe}\ E \mid \mathsf{if}\ E_1\ \mathsf{then}\ E_2\ \mathsf{else}\ E_3 \mid \mathsf{loop}\ E_1\ \mathsf{sum}\ E_2 \\
\text{distribution} & D & ::= & \mathsf{Bernoulli}(\theta) \mid \mathsf{Geo}(\theta) \mid \mathsf{Poisson}(\theta) \\
\text{typing context} & \Gamma & ::= & \varnothing \mid \Gamma, X : T
\end{array}$$

$$\boxed{\vdash D}$$

$$\frac{\theta \in [0,1]}{\vdash \mathsf{Bernoulli}(\theta)} \qquad \frac{\theta \in (0,1]}{\vdash \mathsf{Geo}(\theta)} \qquad \frac{\theta \in (0,\infty)}{\vdash \mathsf{Poisson}(\theta)}$$

$$\boxed{\Gamma \vdash E : T}$$

$$\frac{\Gamma(X) = T}{\Gamma \vdash X : T} \qquad \frac{\forall i, \Gamma(X_i) = \mathbb{N}}{\Gamma \vdash n + \sum_i n_i \cdot X_i : \mathbb{N}} \qquad \frac{\Gamma \vdash E_1 : T_1 \quad \Gamma \vdash E_2 : T_2}{\Gamma \vdash (E_1, E_2) : T_1 \times T_2} \qquad \frac{\Gamma \vdash E : T_1 \times T_2}{\Gamma \vdash \mathsf{fst}\ E : T_1}$$

$$\frac{\Gamma \vdash E : T_1 \times T_2}{\Gamma \vdash \mathsf{snd}\ E : T_2} \qquad \frac{\Gamma \vdash E_1 : T_1 \quad \Gamma, X : T_1 \vdash E_2 : T_2}{\Gamma \vdash \mathsf{let}\ X = E_1\ \mathsf{in}\ E_2 : T_2} \qquad \frac{\Gamma(X) = \mathbb{N} \quad A = \{n_1, \ldots, n_k\}}{\Gamma \vdash X \in A : \mathbb{N}} \qquad \frac{\vdash D}{\Gamma \vdash \mathsf{sample}\ D : \mathbb{N}}$$

$$\frac{\Gamma \vdash E : \mathbb{N}}{\Gamma \vdash \mathsf{observe}\ E : \mathbb{N}} \qquad \frac{\Gamma \vdash E_1 : \mathbb{N} \quad \Gamma \vdash E_2 : T \quad \Gamma \vdash E_3 : T}{\Gamma \vdash \mathsf{if}\ E_1\ \mathsf{then}\ E_2\ \mathsf{else}\ E_3 : T} \qquad \frac{\Gamma \vdash E_1 : \mathbb{N} \quad \varnothing \vdash E_2 : \mathbb{N}}{\Gamma \vdash \mathsf{loop}\ E_1\ \mathsf{sum}\ E_2 : \mathbb{N}}$$

Figure 5. Syntax and type system of PIL.

Geni can handle this example effectively. It is able to compile the program to a GF that represents the exact distribution of the number of successfully transmitted packets. To the best of our knowledge, this is the first time exact solutions can be derived automatically for such a program.

4 PIL: A Probabilistic Intermediate Language

In this section, we formalize the core language PIL, the source language of the GF compilation we will describe in Section 5.

4.1 Syntax and Typing Rules of PIL

Figure 5 shows the syntax and typing rules of PIL.

A compiler intermediate language. PIL is intended as a compiler intermediate language, where the only base type is $\mathbb{N}$.

The Geni surface language supports additional high-level features for user convenience: Booleans, bounded integers, non-recursive function calls, and statically bounded loops. These features can be desugared into PIL. Booleans are encoded as natural numbers, with logical operators expressed using affine natural-number arithmetic and the if-then-else construct; for example, Figure 1 shows an encoding of disjunction. The other features are handled in the same way as in Dice [19]: bounded

integers (i.e., integers of statically known bit widths) and their arithmetic operations are supported via the binary encoding [5], function calls are supported via inlining, and statically bounded loops are supported via unrolling. We do not capture these features in our formalism for simplicity; we will focus on PIL and the compilation from PIL to DIL.

PIL types include the base type $\mathbb{N}$ and product types $T_1 \times T_2$. We overload notations to avoid cluttering the syntax. For example, depending on the context, $\mathbb{N}$ can denote the base type in PIL or the set of natural numbers. Similarly, n can denote a constant expression in PIL or a natural number at the meta-level. Pairs are useful for encoding integers as tuples of binary values.

The expression sample D draws a sample from a distribution D. We use flip θ as a shorthand for sample Bernoulli(θ). The expression observe E conditions the program execution on the value of the subexpression E being nonzero, and the expression if E_1 then E_2 else E_3 tests if E_1 is nonzero and evaluates E_2 or E_3 accordingly.

Expressive power. Not all ways of combining variables are known to have closed-form generating functions. Like prior work [21, 8, 36, 22], we make syntactic restrictions such as allowing only affine operations on variables, viz., $n + \sum_i n_i \cdot X_i$. Despite the restrictions, some non-affine integer operations are expressible via the binary encoding of unsigned integers.

All Dice programs can be expressed in Geni. Moreover, while only finite-support distributions are allowed in Dice, Geni additionally allows infinite-support distributions. Geni also allows some common continuous distributions to be used as priors, though we do not model them here for the simplicity of the formalism; they can be added as an extension of our development and do not pose semantic challenges.

Further, PIL supports a stochastic loop construct loop E_1 sum E_2: the loop executes E_1 times, each time adding the value of executing E_2 to an accumulator. The number of iterations need not be statically known and can be random. The expression E_2 can contain randomness too, but each iteration executes E_2 independently. This independence between loop iterations is reflected in the typing rule, which requires E_2 to be typed under an empty context.

This specialized loop construct is useful for expressing a common pattern found in probabilistic models of evolutionary processes, queuing systems, and communication networks: summing a stochastic number of independent, identically distributed (i.i.d.) random variables. The number of loop iterations is random and unbounded (i.e., it can be arbitrarily large), though the loop eventually terminates after a finite number of iterations. Geni's support of this feature is novel; Genfer [36] does not consider compiling any stochastic loops.

This loop construct subsumes built-in distributions like Binomial(X, θ), NegBinomial(X, θ), and Poisson$(\theta \cdot X)$ in Genfer [36]. They are special cases where the summand is a sample expression:

$$\mathsf{Binomial}(X,\theta) \coloneqq \mathsf{loop}\ X\ \mathsf{sum}\ (\mathsf{sample\ Bernoulli}(\theta)) \tag{4.1}$$

$$\mathsf{NegBinomial}(X,\theta) \coloneqq \mathsf{loop}\ X\ \mathsf{sum}\ (\mathsf{sample\ Geo}(\theta)) \tag{4.2}$$

$$\mathsf{Poisson}(\theta \cdot X) \coloneqq \mathsf{loop}\ X\ \mathsf{sum}\ (\mathsf{sample\ Poisson}(\theta)) \tag{4.3}$$

For example, NegBinomial(X, θ) represents the number of failures before the X-th success in a sequence of Bernoulli trials with success probability θ. It is a generalization of Geo(θ), which represents the number of failures before the *first* success. Hence, NegBinomial(X, θ) can be expressed, with PIL's loop–sum construct, as the sum of X i.i.d. geometric samples.

4.2 Denotational Semantics of PIL

In this section, we give a denotational semantics for PIL. Basics of measure theory are assumed.

Following standard techniques for formally modeling probabilistic programs, we interpret types and typing contexts as measurable spaces:

$$[\![\mathbb{N}]\!] := \mathbb{N} \qquad [\![T_1 \times T_2]\!] := [\![T_1]\!] \otimes [\![T_2]\!] \qquad [\![\Gamma]\!] := \bigotimes_{X \in \mathbf{dom}(\Gamma)} [\![\Gamma(X)]\!]$$

The base type $\mathbb{N}$ is interpreted as the standard Borel space of natural numbers. A product type $T_1 \times T_2$ is interpreted as the product measurable space of the interpretations of T_1 and T_2. A typing context Γ is interpreted as the measurable space of substitutions γ such that for all $X \in \mathbf{dom}(\Gamma)$, $\gamma(X) \in [\![\Gamma(X)]\!]$.

Built-in distributions in PIL are interpreted as probability measures on $\mathbb{N}$. The interpretations are standard. For example, $\mathsf{Poisson}(\theta)$ is interpreted as a measure $[\![\mathsf{Poisson}(\theta)]\!] := \lambda A.\ \sum_{n \in A} e^{-\theta}\theta^n/n!$ that sums the probability masses over all natural numbers in the input set A. We omit the measure denotations for the other built-in distributions.

To give meanings to well-typed expressions, we use the standard monad [15] that assigns to a measurable space S the space $\mathbf{Meas}\ S$ of all measures on S:

$$\mathcal{P}[\![T]\!] := \mathbf{Meas}\ [\![T]\!] \qquad \mathcal{P}[\![\Gamma]\!] := \mathbf{Meas}\ [\![\Gamma]\!]$$

Interpreting well-typed expressions as measures or measure transformers? One way to interpret a well-typed expression $\Gamma \vdash E : T$ is to define a function

$$\mathcal{P}[\![\Gamma \vdash E : T]\!] : [\![\Gamma]\!] \to \mathcal{P}[\![T]\!]. \tag{4.4}$$

For example, in this approach, variables and let-bindings are interpreted as follows:

$$\mathcal{P}[\![\Gamma \vdash X : T]\!]\ \gamma := \mathbf{ret}\ \gamma(X)$$

$$\mathcal{P}[\![\Gamma \vdash \mathsf{let}\ X = E_1\ \mathsf{in}\ E_2 : T_2]\!]\ \gamma := \mathbf{do} \begin{Bmatrix} v \leftarrow \mathcal{P}[\![\Gamma \vdash E_1 : T_1]\!]\ \gamma; \\ \mathcal{P}[\![\Gamma, X : T_1 \vdash E_2 : T_2]\!]\ \gamma[+X \mapsto v] \end{Bmatrix}$$

Here, $\mathbf{ret}\ v$ is the unit of the monad: it lifts a value v to the Dirac measure at v. The $\mathbf{do}\{v \leftarrow \mu;\ f(v)\}$ notation is the bind operation of the monad: it samples a value v from the measure μ and feeds it to f to obtain a new measure. Most denotational semantics of functional PPLs in the literature are defined in this way [32].

We could follow this approach for PIL as well. However, we would like a semantics that is more enlightening for the purpose of GF compilation—that is, a semantics that makes clear the close connection between the measure-theoretic interpretation $\mathcal{P}[\![\text{-}]\!]$ (Figure 6) and the GF compilation $\mathcal{G}[\![\text{-}]\!]$ (Figure 7), so much so that most structural rules in the GF compilation can be directly adapted from the denotational semantics.

The standard interpretation (4.4) does not seem to serve this purpose well. $\mathcal{P}[\![T]\!]$ in (4.4) is a space of *measures*, which corresponds to the space of generating functions of measures on $[\![T]\!]$. But $[\![\Gamma]\!]$ in (4.4) is not a space of measures; it does not have a direct correspondence in the GF world.

This observation leads us to define a semantics that interprets a well-typed expression $\Gamma \vdash E : T$ as a function with domain $\mathcal{P}[\![\Gamma]\!]$ and codomain $\forall Y \notin \mathbf{dom}(\Gamma), \mathcal{P}[\![\Gamma, Y : T]\!]$:

$$\mathcal{P}[\![\Gamma \vdash E : T]\!] : \mathcal{P}[\![\Gamma]\!] \to \forall Y \notin \mathbf{dom}(\Gamma), \mathcal{P}[\![\Gamma, Y : T]\!]. \tag{4.5}$$

The codomain is itself a space of (dependently typed) functions, taking a fresh variable Y as input and producing a measure in the space $\mathcal{P}[\![\Gamma, Y : T]\!]$. The fresh variable Y represents the result of evaluating E. The denotation $\mathcal{P}[\![\Gamma \vdash E : T]\!]$ is essentially a *measure transformer*: given an input measure $\mu \in \mathcal{P}[\![\Gamma]\!]$ and a variable Y, the output $\mathcal{P}[\![\Gamma \vdash E : T]\!]\ \mu\ Y$ is a measure that characterizes the joint (unnormalized) distribution of Y and the variables in $\mathbf{dom}(\Gamma)$.

Interpreting a closed program $\varnothing \vdash E : T$ then amounts to a tree traversal that successively applies the measure transformers denoting the subexpressions, starting from a measure $\mu_\varnothing \in \mathcal{P}[\![\varnothing]\!]$ and concluding with a measure in $\mathcal{P}[\![Y : T]\!]$. Here, $\mu_\varnothing$ is the measure that assigns probability 1 to the empty context, and Y is a fresh variable representing the result of evaluating the program.

As will become clear in Section 5, this interpretation into measure transformers lends itself to elucidating the connection to GF compilation.

Interpreting well-typed expressions as measure transformers. Figure 6 defines the denotational semantics of PIL expressions. In the meta-language, in addition to the usual monadic operations, we use the following notations:

- For $\gamma \in [\![\Gamma]\!]$ and $X \notin \mathbf{dom}(\Gamma)$, the notation $\gamma[+X \mapsto v]$ is the extension of γ with X mapped to v. For $\gamma \in [\![\Gamma]\!]$ and $X \in \mathbf{dom}(\Gamma)$, the notation $\gamma[\text{-}X]$ is the elimination of X from $\mathbf{dom}(\gamma)$, while $\gamma[X \mapsto v]$ is the update of γ with X mapped to v.
- For $b \in \{\top, \bot\}$ and $\mu \in \mathbf{Meas}\ S$, the notation $\mathbf{condition}\ b;\ \mu$ is defined as $\mathbf{if}\ b\ \mathbf{then}\ \mu\ \mathbf{else}\ \lambda_.\,0$. It restricts the measure μ to the event b. Here, $\lambda_.\,0$ is the null measure.
- $\mathbf{fold}\ :\ (R \to S \to S) \to S \to \mathbf{List}\ R \to S$ is the usual fold operation on lists.
- For $\mu \in \mathbf{Meas}\ S$, if the normalization constant $\mu([\![S]\!])$ is not zero, then we define $\mathbf{normalize}\ \mu$ as the normalized measure $\mu/\mu([\![S]\!])$.

The definitions in Figure 6 are straightforward for the most part, by induction on typing derivations. For example, a let binding $\Gamma \vdash \mathsf{let}\ X = E_1\ \mathsf{in}\ E_2\ :\ T_2$ is interpreted as the composition of the measure transformer denoting E_1 under Γ and the measure transformer denoting E_2 under the extended context $\Gamma, X\ :\ T_1$. The final measure, of type $\mathbf{Meas}\ [\![\Gamma, Y\ :\ T_2]\!]$, is obtained by marginalizing out the variable X, since X is in scope only within E_2. The definitions of these inductive cases will directly inform the corresponding rules in the GF compilation.

To interpret observe and if expressions, we use an auxiliary definition

$$\mathcal{P}(\!(\Gamma \vdash E\ :\ \mathbb{N})\!)\ :\ \mathcal{P}[\![\Gamma]\!] \to \{\bot, \top\} \to \mathcal{P}[\![\Gamma]\!]. \tag{4.6}$$

Given an input measure $\mu \in \mathcal{P}[\![\Gamma]\!]$, for $b \in \{\bot, \top\}$, the output measure $\mathcal{P}(\!(\Gamma \vdash E\ :\ \mathbb{N})\!)\ \mu\ b$ is the restriction of μ to those substitutions for Γ where E evaluates to zero or nonzero, respectively.

The interpretation $\mathcal{P}[\![\Gamma \vdash \mathsf{loop}\ E_1\ \mathsf{sum}\ E_2\ :\ \mathbb{N}]\!]\ \mu\, Y$ of the expression $\mathsf{loop}\ E_1\ \mathsf{sum}\ E_2$ is a bit more involved. It is obtained by folding a function $\lambda_.\,f\ :\ \mathbb{N} \to \mathbf{Meas}\ [\![\Gamma, Y\ :\ \mathbb{N}]\!] \to \mathbf{Meas}\ [\![\Gamma, Y\ :\ \mathbb{N}]\!]$ over the list $[1, \ldots, \gamma(X)] \in \mathbf{List}\ \mathbb{N}$. The fresh variable X stands for the result of evaluating E_1, namely the number of iterations. The function $\lambda_.\,f$ adds the result of evaluating E_2 to the accumulator. It ignores the loop index, as the evaluation of E_2 is i.i.d. across iterations.

For a closed program E, its denotation $\mathcal{P}[\![\varnothing \vdash E\ :\ T]\!]\ \mu_\varnothing\ Y$ is an unnormalized measure, if E uses Bayesian conditioning. The normalized measure is obtained by $\mathbf{normalize}\ (\mathcal{P}[\![\varnothing \vdash E\ :\ T]\!]\ \mu_\varnothing\ Y)$.

We will abbreviate $\mathcal{P}[\![\Gamma \vdash E\ :\ T]\!]\ \mu\, Y$ as $\mathcal{P}[\![E]\!]\ \mu\, Y$ when Γ and T are clear from the context.

5 GF Compilation

In this section, we formally define the translation from PIL to DIL.

5.1 Target Language DIL

The target language DIL provides constructs that can be used to express generating functions. While the base type in PIL is $\mathbb{N}$, the base type in DIL is $\mathbb{R}$. The syntax of DIL is defined as follows:

$$\begin{array}{rrcl}
\text{type} & \tau & ::= & \mathbb{R} \mid \tau \times \tau \\
\text{expression} & e & ::= & x \mid n \mid \theta \mid e_1 + e_2 \mid e_1 \odot e_2 \mid e^n \mid \mathsf{exp}\ e \mid 1/e \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid \\
& & & (e_1, e_2) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \mathsf{D}^n_{x=\theta}\, e \\
\text{typing context} & \Xi & ::= & \varnothing \mid \Xi, x\ :\ \tau
\end{array}$$

$$\boxed{\mathcal{P}[\![\Gamma \vdash E : T]\!] : \mathcal{P}[\![\Gamma]\!] \to \forall Y \notin \mathbf{dom}(\Gamma), \mathcal{P}[\![\Gamma, Y : T]\!]}$$

$$\mathcal{P}[\![\Gamma \vdash X : T]\!]\ \mu\, Y \coloneqq \mathbf{do}\left\{ \gamma \leftarrow \mu;\ \mathbf{ret}\ \gamma[+Y \mapsto \gamma(X)] \right\}$$

$$\mathcal{P}\left[\!\!\left[\Gamma \vdash n + \textstyle\sum_i n_i \cdot X_i : \mathbb{N}\right]\!\!\right] \mu\, Y \coloneqq \mathbf{do}\left\{ \gamma \leftarrow \mu;\ \mathbf{ret}\ \gamma[+Y \mapsto n + \textstyle\sum_i n_i \cdot \gamma(X_i)] \right\}$$

$$\mathcal{P}[\![\Gamma \vdash (E_1, E_2) : T_1 \times T_2]\!]\ \mu\, Y \coloneqq \mathbf{do}\left\{ \begin{array}{l} \gamma \leftarrow \mathcal{P}[\![\Gamma, _ : T_1 \vdash E_2 : T_2]\!]\ \left(\mathcal{P}[\![\Gamma \vdash E_1 : T_1]\!]\ \mu\, Y_1\right) Y_2; \\ \mathbf{ret}\ \gamma[+Y \mapsto (\gamma(Y_1), \gamma(Y_2)), \text{-}Y_1, \text{-}Y_2] \end{array} \right\}$$

$$\mathcal{P}[\![\Gamma \vdash \mathsf{fst}\ E : T_1]\!]\ \mu\, Y_1 \coloneqq \mathbf{do}\left\{ \gamma \leftarrow \mathcal{P}[\![\Gamma \vdash E : T_1 \times T_2]\!]\ \mu\, Y;\ \mathbf{ret}\ \gamma[+Y_1 \mapsto \mathbf{fst}\ \gamma(Y), \text{-}Y] \right\}$$

$$\mathcal{P}[\![\Gamma \vdash \mathsf{snd}\ E : T_2]\!]\ \mu\, Y_2 \coloneqq \mathbf{do}\left\{ \gamma \leftarrow \mathcal{P}[\![\Gamma \vdash E : T_1 \times T_2]\!]\ \mu\, Y;\ \mathbf{ret}\ \gamma[+Y_2 \mapsto \mathbf{snd}\ \gamma(Y), \text{-}Y] \right\}$$

$$\mathcal{P}[\![\Gamma \vdash \mathsf{let}\ X = E_1\ \mathsf{in}\ E_2 : T_2]\!]\ \mu\, Y \coloneqq \mathbf{do}\left\{ \begin{array}{l} \gamma \leftarrow \mathcal{P}[\![\Gamma, X : T_1 \vdash E_2 : T_2]\!]\ \left(\mathcal{P}[\![\Gamma \vdash E_1 : T_1]\!]\ \mu\, X\right) Y; \\ \mathbf{ret}\ \gamma[\text{-}X] \end{array} \right\}$$

$$\mathcal{P}[\![\Gamma \vdash X \in A : \mathbb{N}]\!]\ \mu\, Y \coloneqq \mathbf{do}\left\{ \gamma \leftarrow \mu;\ \mathbf{ret}\ \gamma[+Y \mapsto \mathbf{if}\ \gamma(X) \in A\ \mathbf{then}\ 1\ \mathbf{else}\ 0] \right\}$$

$$\mathcal{P}[\![\Gamma \vdash \mathsf{sample}\ D : T]\!]\ \mu\, Y \coloneqq \mathbf{do}\left\{ \gamma \leftarrow \mu;\ v \leftarrow [\![D]\!];\ \mathbf{ret}\ \gamma[+Y \mapsto v] \right\}$$

$$\mathcal{P}[\![\Gamma \vdash \mathsf{observe}\ E : \mathbb{N}]\!]\ \mu\, Y \coloneqq \mathbf{do}\left\{ \gamma \leftarrow \mathcal{P}(\!|\Gamma \vdash E : \mathbb{N}|\!)\ \mu\, \top;\ \mathbf{ret}\ \gamma[+Y \mapsto 0] \right\}$$

$$\mathcal{P}[\![\Gamma \vdash \mathsf{if}\ E\ \mathsf{then}\ E_\top\ \mathsf{else}\ E_\bot : T]\!]\ \mu\, Y \coloneqq \sum_{b \in \{\top, \bot\}} \mathcal{P}[\![\Gamma \vdash E_b : T]\!]\ \left(\mathcal{P}(\!|\Gamma \vdash E : \mathbb{N}|\!)\ \mu\, b\right) Y$$

$$\begin{aligned} \mathcal{P}[\![\Gamma \vdash \mathsf{loop}\ E_1\ \mathsf{sum}\ E_2 : \mathbb{N}]\!]\ \mu\, Y \coloneqq\ & \mathbf{let}\ \mu' = \mathcal{P}[\![\varnothing \vdash E_2 : \mathbb{N}]\!]\ \mu_\varnothing\, Y\ \mathbf{in} \\ & \mathbf{let}\ f = \lambda\mu.\, \mathbf{do}\left\{ \gamma \leftarrow \mu;\ \gamma' \leftarrow \mu';\ \mathbf{ret}\ \gamma\big[Y \mapsto \gamma(Y) + \gamma'(Y)\big] \right\}\mathbf{in} \\ & \mathbf{let}\ \mu_0 = \mathbf{do}\left\{ \gamma \leftarrow \mu;\ \mathbf{ret}\ \gamma[+Y \mapsto 0] \right\}\mathbf{in} \\ & \gamma \leftarrow \mathcal{P}[\![\Gamma \vdash E_1 : \mathbb{N}]\!]\ \mu\, X;\ \mathbf{fold}\ (\lambda_.\, f)\ \mu_0\ [1, \ldots, \gamma(X)] \end{aligned}$$

$$\boxed{\mathcal{P}(\!|\Gamma \vdash E : \mathbb{N}|\!) : \mathcal{P}[\![\Gamma]\!] \to \{\bot, \top\} \to \mathcal{P}[\![\Gamma]\!]}$$

$$\mathcal{P}(\!|\Gamma \vdash E : \mathbb{N}|\!)\ \mu\, \bot \coloneqq \mathbf{do}\left\{ \gamma \leftarrow \mathcal{P}[\![\Gamma \vdash E : \mathbb{N}]\!]\ \mu\, Y;\ \mathbf{condition}\ \gamma(y) = 0;\ \mathbf{ret}\ \gamma \right\}$$

$$\mathcal{P}(\!|\Gamma \vdash E : \mathbb{N}|\!)\ \mu\, \top \coloneqq \mathbf{do}\left\{ \gamma \leftarrow \mathcal{P}[\![\Gamma \vdash E : \mathbb{N}]\!]\ \mu\, Y;\ \mathbf{condition}\ \gamma(y) \neq 0;\ \mathbf{ret}\ \gamma \right\}$$

Figure 6. Denotational semantics of PIL: interpreting expressions.

DIL is in large part a standard expression language. We will use the notation $\mathcal{D}[\![\Xi \vdash e : \tau]\!]\, \xi$ for the denotational meaning of a well-typed expression e:

$$\mathcal{D}[\![\Xi \vdash e : \tau]\!] : [\![\Xi]\!] \to [\![\tau]\!]. \tag{5.1}$$

It interprets an open expression e as a function of the substitution ξ for the variables bound in Ξ.

The construct $e_1 \odot e_2$ is the component-wise multiplication, with the following typing rules, one for scalars and one for pairs:

$$\frac{\Xi \vdash e_1 : \mathbb{R} \quad \Xi \vdash e_2 : \mathbb{R}}{\Xi \vdash e_1 \odot e_2 : \mathbb{R}} \qquad \frac{\Xi \vdash e_1 : \tau_1 \times \tau_2 \quad \Xi \vdash e_2 : \tau_1 \times \tau_2}{\Xi \vdash e_1 \odot e_2 : \tau_1 \times \tau_2}$$

We will use $e_1 \cdot e_2$ to mean $e_1 \odot e_2$ when e_1 and e_2 are scalars (i.e., when they have type $\mathbb{R}$).

The construct $\mathsf{D}^n_{x=\theta}\, e$ expresses the n-th derivative of e with respect to x at point θ:

$$\mathcal{D}[\![\Xi \vdash \mathsf{D}^n_{x=\theta}\, e : \mathbb{R}]\!]\, \xi := \left(\frac{\mathrm{d}^n}{\mathrm{d}v^n}\, \lambda v.\, \mathcal{D}[\![\Xi, x : \mathbb{R} \vdash e : \mathbb{R}]\!]\, \xi[+x \mapsto v] \right)_{v=\theta}. \tag{5.2}$$

In this denotational interpretation, we take the n-th derivative of the denotation of the subexpression e with respect to a metalevel variable v and evaluate the derivative at the point $v = \theta$.

The interpretations of the other constructs are standard and thus omitted. We will abbreviate $\mathcal{D}[\![\Xi \vdash e : \tau]\!]\, \xi$ as $\mathcal{D}[\![e]\!]\, \xi$ when Ξ and τ are clear from the context.

We will use $\mathsf{let}\ (x_1, x_2) = x\ \mathsf{in}\ e$ as a shorthand for $\mathsf{let}\ x_1 = \mathsf{fst}\ x\ \mathsf{in}\ \mathsf{let}\ x_2 = \mathsf{snd}\ x\ \mathsf{in}\ e$. Capture-avoiding substitution of e' for x in e is notated by $e[x \mapsto e']$.

5.2 Generalized Generating Functions

We now generalize the idea of generating functions to **Meas** $[\![\Gamma]\!]$, i.e., measures over $[\![\Gamma]\!]$.

First, we define a translation $|\text{-}|$ from PIL types and PIL typing contexts to those in DIL:

$$|\mathbb{N}| := \mathbb{R} \qquad |T_1 \times T_2| := |T_1| \times |T_2| \qquad |\varnothing| = \varnothing \qquad |\Gamma, X : T| = |\Gamma|, x : |T|$$

Notice that a variable name X in PIL is mapped to the variable name x in DIL, by analogy with the convention that uppercase letters denote random variables and lowercase letters denote their GFs' formal parameters (Section 2).

For a measure $\mu \in$ **Meas** $[\![\Gamma]\!]$, we define its GF $\mathbb{G}_\mu$ as a function of type $[\![|\Gamma|]\!] \to \mathbb{R}$:

$$\mathbb{G}_\mu(\xi) := \mathbb{E}_{\gamma \sim \mu}[\xi^\gamma] = \int_{[\![\Gamma]\!]} \xi^\gamma\, \mu(\mathrm{d}\gamma). \tag{5.3}$$

Here, we have generalized GFs to allow for substitutions $\xi \in [\![|\Gamma|]\!]$ as inputs, using Lebesgue integration with respect to the measure $\mu \in$ **Meas** $[\![\Gamma]\!]$. The notation ξ^γ is defined as

$$\xi^\gamma := \prod_{X \in \mathbf{dom}(\Gamma)} \xi(x)^{\gamma(X)}, \tag{5.4}$$

by analogy with how $\boldsymbol{x}^{\boldsymbol{X}}$ is defined as $\prod_i x_i^{X_i}$ in the context of vectors (2.5).

When $\Gamma = \varnothing$, we consider the right-hand side of (5.4) to be 1. Thus, $\mathbb{G}_{\mu_\varnothing}$ is the constant function 1.

5.3 Compiling Well-Typed PIL Expressions to GF Transformers

We interpret a PIL typing context Γ into the set of DIL expressions closed under $|\Gamma|$:

$$\mathcal{G}[\![\Gamma]\!] := \left\{ e \,\middle|\, |\Gamma| \vdash e : \mathbb{R} \right\}. \tag{5.5}$$

The denotation of an expression $e \in \mathcal{G}[\![\Gamma]\!]$ is a function of the type $[\![|\Gamma|]\!] \to \mathbb{R}$, per (5.1). Notice that $[\![|\Gamma|]\!] \to \mathbb{R}$ is exactly the type of generalized GFs defined in (5.3).

A well-typed expression $\Gamma \vdash E : T$ in PIL is then compiled by interpreting it as a *GF transformer*:

$$\mathcal{G}[\![\Gamma \vdash E : T]\!] : \mathcal{G}[\![\Gamma]\!] \to \forall y \notin \mathbf{dom}(|\Gamma|), \mathcal{G}[\![\Gamma, Y : T]\!]. \tag{5.6}$$

Notice how the GF compilation in (5.6) is set up in a way that mirrors the measure-transformer semantics in (4.5). Given an input expression e representing the GF of a measure in **Meas** $[\![\Gamma]\!]$, and given a fresh variable y, the output $\mathcal{G}[\![\Gamma \vdash E : T]\!]\ e\ y$ is a DIL expression that represents the GF of the joint distribution of Y and the variables in $\mathbf{dom}(\Gamma)$. We will call the inputs and outputs of these GF transformers *GF expressions*.

Compiling a closed program $\emptyset \vdash E : T$ then amounts to a tree traversal that successively applies the GF transformers to GF expressions, starting from the GF expression $1 \in \mathcal{G}[\![\emptyset]\!]$ and concluding with a GF expression in $\mathcal{G}[\![Y : T]\!]$, where Y is a fresh variable representing the result of evaluating the program. This compilation can also be seen as a symbolic execution, where the intermediate GF expressions are symbolic representations of the intermediate measures obtained by the measure transformers in the denotational semantics.

Figure 7 defines the GF compilation of well-typed PIL expressions, by induction on typing derivations.

All the inductive cases, except for the loop–sum construct, are almost directly read off from the denotational semantics in Figure 6. For example, a let binding $\Gamma \vdash \mathsf{let}\ X = E_1\ \mathsf{in}\ E_2 : T_2$ is compiled by composing the GF transformer denoting E_1 under Γ and the GF transformer denoting E_2 under the extended context $\Gamma, X : T_1$. The final GF expression, of type $\mathcal{G}[\![\Gamma, Y : T_2]\!]$, is obtained by marginalizing out the variable X, since X is in scope only within E_2. Following (2.6), the marginalization is done by setting x to the multiplicative identity 1. Since X can have a product type, we set x to $\mathbf{1}_{T_1}$, defined as follows:

$$\mathbf{1}_{\mathbb{N}} := 1 \qquad\qquad \mathbf{1}_{T_1 \times T_2} := (\mathbf{1}_{T_1}, \mathbf{1}_{T_2})$$

The interpretations of observe and if expressions use an auxiliary definition

$$\mathcal{G}(\!|\Gamma \vdash E : \mathbb{N}|\!) : \mathcal{G}[\![\Gamma]\!] \to \{\bot, \top\} \to \mathcal{G}[\![\Gamma]\!]. \tag{5.7}$$

Given an input GF expression $e \in \mathcal{G}[\![\Gamma]\!]$, for $b \in \{\bot, \top\}$, the output GF expression $\mathcal{G}(\!|\Gamma \vdash E : \mathbb{N}|\!)\ e\ b$ is the restriction of e to those substitutions for Γ that cause E to evaluate to zero or nonzero, respectively. Importantly, differentiation is not needed: following (3.4), the GF expression for the zero event is obtained by evaluating $\mathcal{G}[\![\Gamma \vdash E : \mathbb{N}]\!]\ e\ y$ at $y = 0$, and the GF expression for the non-zero event is obtained by subtracting the zero event from the full event.

The base cases in Figure 7 follow from standard results about generating functions.

- The case of affine transformations is a direct application of (2.7).
- The cases of the built-in distributions follow from (2.2) and (2.10) as well as (2.9). Notice that in the case of sample Geo(θ), the GF expression may never lead to division by zero: the type system ensures that $\theta \in (0, 1]$, and the GF compilation ensures that the run-time value of y is in $[0, 1]$.
- The case of set membership ($X \in A$) applies (2.3) to extract the masses of the elements in A via higher-order derivatives.
- The case of variables is defined with component-wise multiplication $\odot$; when the variable has the base type $\mathbb{N}$, it reduces to the affine-transformation case.

Now, consider the loop E_1 sum E_2 construct. While its measure semantics in Figure 6 takes some space to define, its GF-transformer interpretation is pleasantly concise—it is essentially composing two GFs. To gain some intuition, consider the case where $E_1 = X$ for some X sampled from some distribution D_1 and $E_2 = \mathsf{sample}\ D_2$ for some distribution D_2:

$$\mathsf{let}\ X = \mathsf{sample}\ D_1\ \mathsf{in}\ \mathsf{loop}\ X\ \mathsf{sum}\ (\mathsf{sample}\ D_2)$$

$$\boxed{\mathcal{G}[\![\Gamma \vdash E : T]\!] : \mathcal{G}[\![\Gamma]\!] \to \forall y \notin \mathbf{dom}(|\Gamma|), \mathcal{G}[\![\Gamma, Y : T]\!]}$$

$$\mathcal{G}[\![\Gamma \vdash X : T]\!]\, e\, y \coloneqq e[x \mapsto x \odot y]$$

$$\mathcal{G}\left[\!\!\left[\Gamma \vdash n + \textstyle\sum_i n_i \cdot X_i : \mathbb{N}\right]\!\!\right] e\, y \coloneqq y^n \cdot e[\ldots,\ x_i \mapsto x_i \cdot y^{n_i},\ \ldots]$$

$$\begin{aligned}\mathcal{G}[\![\Gamma \vdash (E_1, E_2) : T_1 \times T_2]\!]\, e\, y \coloneqq{} & \mathsf{let}\ (y_1, y_2) = y\ \mathsf{in} \\ & \mathcal{G}[\![\Gamma, _ : T_1 \vdash E_2 : T_2]\!]\, \big(\mathcal{G}[\![\Gamma \vdash E_1 : T_1]\!]\, e\, y_1\big)\, y_2\end{aligned}$$

$$\mathcal{G}[\![\Gamma \vdash \mathsf{fst}\ E : T_1]\!]\, e\, y_1 \coloneqq \mathsf{let}\ y = (y_1, \mathbf{1}_{T_2})\ \mathsf{in}\ \mathcal{G}[\![\Gamma \vdash E : T_1 \times T_2]\!]\, e\, y$$

$$\mathcal{G}[\![\Gamma \vdash \mathsf{snd}\ E : T_2]\!]\, e\, y_2 \coloneqq \mathsf{let}\ y = (\mathbf{1}_{T_1}, y_2)\ \mathsf{in}\ \mathcal{G}[\![\Gamma \vdash E : T_1 \times T_2]\!]\, e\, y$$

$$\begin{aligned}\mathcal{G}[\![\Gamma \vdash \mathsf{let}\ X = E_1\ \mathsf{in}\ E_2 : T_2]\!]\, e\, y \coloneqq{} & \mathsf{let}\ x = \mathbf{1}_{T_1}\ \mathsf{in} \\ & \mathcal{G}[\![\Gamma, X : T_1 \vdash E_2 : T_2]\!]\, \big(\mathcal{G}[\![\Gamma \vdash E_1 : T_1]\!]\, e\, x\big)\, y\end{aligned}$$

$$\mathcal{G}[\![\Gamma \vdash X \in A : \mathbb{N}]\!]\, e\, y \coloneqq \mathsf{let}\ z = \sum_{n \in A} \frac{\mathsf{D}^n_{x=0}\, e}{n!} \cdot x^n\ \mathsf{in}\ e - z + y \cdot z$$

$$\mathcal{G}[\![\Gamma \vdash \mathsf{sample\ Bernoulli}(\theta) : \mathbb{N}]\!]\, e\, y \coloneqq e \cdot (1 - \theta + \theta \cdot y)$$

$$\mathcal{G}[\![\Gamma \vdash \mathsf{sample\ Poisson}(\theta) : \mathbb{N}]\!]\, e\, y \coloneqq e \cdot (\mathsf{exp}\ \theta \cdot (y - 1))$$

$$\mathcal{G}[\![\Gamma \vdash \mathsf{sample\ Geo}(\theta) : \mathbb{N}]\!]\, e\, y \coloneqq e \cdot \frac{\theta}{1 - (1 - \theta) \cdot y}$$

$$\mathcal{G}[\![\Gamma \vdash \mathsf{observe}\ E : \mathbb{N}]\!]\, e\, y \coloneqq (\mathcal{G}(\!|\Gamma \vdash E : \mathbb{N}|\!)\, e\, \top) \cdot y^0$$

$$\mathcal{G}[\![\Gamma \vdash \mathsf{if}\ E\ \mathsf{then}\ E_\top\ \mathsf{else}\ E_\bot : T]\!]\, e\, y \coloneqq \sum_{b \in \{\top, \bot\}} \mathcal{G}[\![\Gamma \vdash E_b : T]\!]\, \big(\mathcal{G}(\!|\Gamma \vdash E : \mathbb{N}|\!)\, e\, b\big)\, y$$

$$\mathcal{G}[\![\Gamma \vdash \mathsf{loop}\ E_1\ \mathsf{sum}\ E_2 : \mathbb{N}]\!]\, e\, y \coloneqq \mathsf{let}\ x = \mathcal{G}[\![\varnothing \vdash E_2 : \mathbb{N}]\!]\, 1\, y\ \mathsf{in}\ \mathcal{G}[\![\Gamma \vdash E_1 : \mathbb{N}]\!]\, e\, x$$

$$\boxed{\mathcal{G}(\!|\Gamma \vdash E : \mathbb{N}|\!) : \mathcal{G}[\![\Gamma]\!] \to \{\bot, \top\} \to \mathcal{G}[\![\Gamma]\!]}$$

$$\mathcal{G}(\!|\Gamma \vdash E : \mathbb{N}|\!)\, e\, \bot \coloneqq \mathsf{let}\ y = 0\ \mathsf{in}\ \mathcal{G}[\![\Gamma \vdash E : \mathbb{N}]\!]\, e\, y$$

$$\mathcal{G}(\!|\Gamma \vdash E : \mathbb{N}|\!)\, e\, \top \coloneqq \big(\mathsf{let}\ y = 1\ \mathsf{in}\ \mathcal{G}[\![\Gamma \vdash E : \mathbb{N}]\!]\, e\, y\big) - \mathcal{G}(\!|\Gamma \vdash E : \mathbb{N}|\!)\, e\, \bot$$

Figure 7. Compiling PIL to DIL: Interpreting PIL expressions as GF transformers.

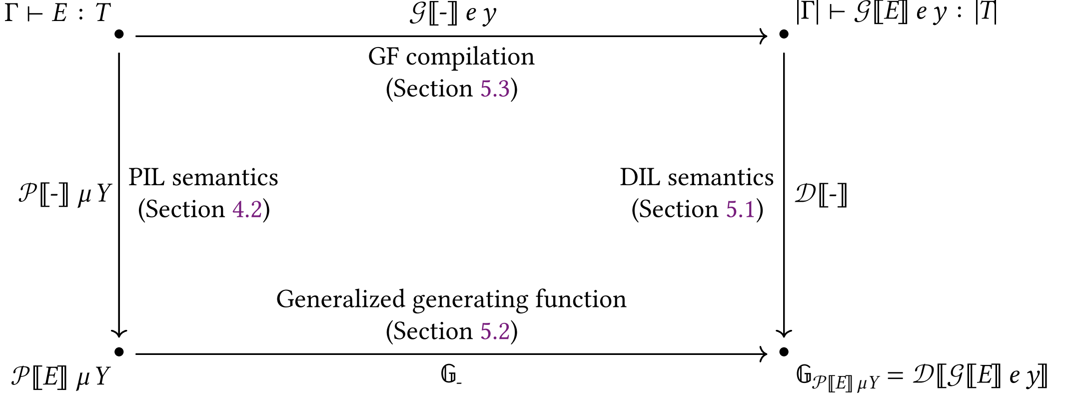


Figure 8. GF compilation is correct if, given $\mathcal{D}[\![e]\!] = \mathbb{G}_\mu$, the diagram commutes (Theorem 5.1).

Suppose D_1 and D_2 have GFs $\mathbb{G}_X$ and $\mathbb{G}_Y$. Let Z be the random variable representing the sum of X i.i.d. samples of D_2. Then by (2.8), we have that the GF of Z is $\mathbb{G}_Y(z)^X$. But X is a random variable—we need to marginalize it out. So the GF of Z is actually

$$\mathbb{G}_Z(z) = \sum_{n\in\mathbb{N}} \mathbb{P}[X = n] \cdot \mathbb{G}_Y(z)^X = \mathbb{E}_X\Big[\mathbb{G}_Y(z)^X\Big] = \mathbb{G}_X(\mathbb{G}_Y(z)) = (\mathbb{G}_X \circ \mathbb{G}_Y)(z),$$

which is the composition of the GFs of D_1 and D_2.

For a closed program E, the compiled DIL program $e := \mathcal{G}[\![\varnothing \vdash E : T]\!]\ 1\ y$ stands for an unnormalized GF, if E uses Bayesian conditioning. The normalized GF is represented by the DIL expression $e/e[y \mapsto \mathbf{1}_T]$.

We will abbreviate $\mathcal{G}[\![\Gamma \vdash E : T]\!]\ e\ y$ as $\mathcal{G}[\![E]\!]\ e\ y$ when Γ and T are clear from the context.

5.4 Compiler Correctness

A key theorem is that the GF compilation presented in Figure 7 is semantics-preserving.

Theorem 5.1 (GF compilation preserves semantics). *Suppose* $\Gamma \vdash E : T$, $\mu \in \mathcal{P}[\![\Gamma]\!]$, $e \in \mathcal{G}[\![\Gamma]\!]$, $\mathcal{D}[\![e]\!] = \mathbb{G}_\mu$, *and* $Y \notin \mathbf{dom}(\Gamma)$. *Then*

$$\mathbb{G}_{\mathcal{P}[\![E]\!]\ \mu Y} = \mathcal{D}[\![\mathcal{G}[\![E]\!]\ e\ y]\!].$$

That is, the diagram in Figure 8 commutes.

The proof of Theorem 5.1 can be found in an appendix in the extended version [28]. Thanks to the close correspondence between the measure semantics in Figure 6 and the GF compilation in Figure 7, the proof is largely straightforward.

6 Implementation

Geni is implemented in Rust. The Geni system consists of a front end that lowers Geni programs to PIL, a middle end that compiles PIL programs to DIL, and a back end that interprets DIL programs.

Middle end. The implementation of the middle-end compiler includes simple optimizations not covered in the formalization. In particular, it performs live variable analysis, so that the compilation can eagerly marginalize out variables that are no longer needed, to reduce the size of the GF expressions.

Back ends. For good performance of the DIL interpreter, it is important that results of evaluating intermediate GF expressions be memoized. As Figure 7 shows, the translation of conditional expressions duplicates GF expressions, which could lead to exponential blowup for programs with a large number of conditionals.

For evaluating higher-order derivatives, we use the same framework for automatic differentiation that Zaiser et al. [36] developed for Genfer, which works by computing the coefficients of the Taylor expansion of the generating function.

Generating functions, compared to other representations such as BDDs, offer an advantage: because generating functions consist primarily of pure code performing arithmetic operations, they can be easily compiled to any general-purpose programming language, especially when differentiation is not needed. This insight suggests that Geni can have *compiler* back ends.

Hence, in addition to the interpreter for DIL, we implement a simple compiler that translates differentiation-free DIL programs to C code, which in turn can be compiled to fast native code using an off-the-shelf optimizing C compiler like GCC or Clang. Even though the standard C compilers do not support differentiation, Geni's C back end is applicable to a large class of programs (including all programs expressible by Dice), as the design of Geni reduces the need for differentiation. This design choice uniquely positions Geni to take advantage of the performance potential of generating functions, which prior work does not tap into [21, 8, 36, 22].

As expected, the generated C code is much faster than the interpreter. Nevertheless, we use the interpreter in the experiments (Section 7) for a fair comparison with Dice and Genfer: weighted model counting on BDDs is an interpreted rather than compiled procedure, and Genfer evaluates GFs using an interpreter. In the experiments, we measure the combined time of compiling to BDDs (resp. GFs) and interpreting the BDDs (resp. GFs).

Continuous priors. Our implementation of Geni allows certain continuous distributions to be used as *priors*—that is, their parameters must be specified at compile time. These distributions include the uniform distribution on an interval $\mathsf{Uniform}(a, b)$, the gamma distribution $\mathsf{Gamma}(\alpha, \beta)$, the exponential distribution $\mathsf{Exp}(\lambda)$, and the normal distribution $\mathsf{Normal}(\mu, \sigma)$. Continuous distributions are compiled using moment-generating functions, as in Genfer [36], for numerical stability.

7 Evaluation

In this section, we evaluate the implementation of Geni on a set of exact-inference benchmarks.

All experiments in this paper were run on a workstation with a 3.6GHz CPU and 128 GB of RAM. The reported times were measured using the Hyperfine command-line tool [20], which automatically performs multiple runs and statistical analysis across the runs.

We focus on comparing the performance of Geni to that of Dice and Genfer. We do not run other exact-inference tools like Psi [14] and Hakaru [29], as previous work [19, 36] has shown that they either cannot express the benchmarks or are generally less scalable than Dice and Genfer.

Probabilistic model checking of finite-horizon Markov chains. Recent work [18] applies exact probabilistic inference to the symbolic verification of Markov chains with respect to finite-horizon reachability properties. The tool, called Rubicon, translates discrete-time Markov chains expressed in the PRISM language [24] to Dice programs and then solves the verification problem by running exact inference using Dice. It is shown that using Dice as a verification back end improves scalability: it enables verification to scale to problem sizes that are infeasible with state-of-the-art probabilistic model checkers like Storm [17].

We compare Dice and Geni on a motivating example from the Rubicon paper [18]. In this Markov model, there are n factories. On any given day, each factory is in one of two states, striking and not striking, and has its own probabilities of beginning striking and stopping a strike. The probability

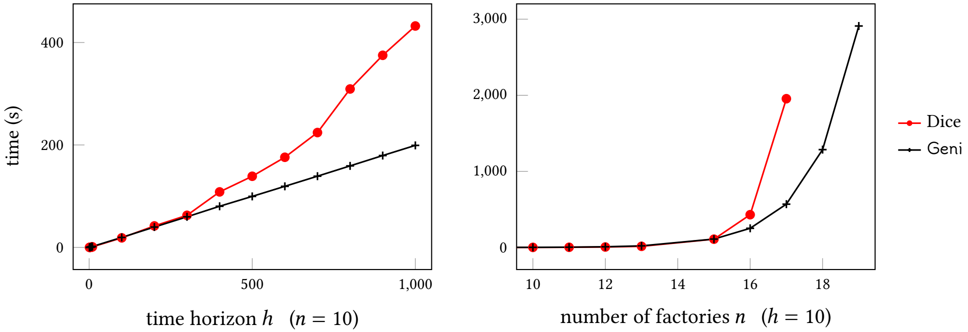


Figure 9. Scaling of Dice and Geni on a model-checking benchmark from Holtzen et al. [18]. In the right diagram, Dice times out for $n \geq 18$.

that each factory goes on strike depends on a common random event: whether it is raining. The probability of rain further depends on the previous day's weather. The verification problem queries the probability that all factories are simultaneously striking within h days.

Given a PRISM description of this model, we generate a Geni program. We measure the time taken by Dice and Geni to solve the verification problem specified by the generated programs. We do not include Genfer in this comparison, as it fails to yield results for relatively small problem sizes.

We assess scalability along two dimensions: the time horizon h and the number of factories n. The latter presents a greater challenge for model checking, as the state space grows exponentially with n but only linearly with h.

- **Scaling with the time horizon h.** We fix $n = 10$ and vary h. Since n is fixed, this experiment examines how well the systems can exploit the Markov property—a type of independence structure—of the Markov chain to achieve computational savings as the time horizon increases. The left diagram in Figure 9 shows how the running time scales with h. Dice exhibits super-linear scaling, while Geni shows linear scaling.
- **Scaling with the number of factories n.** We fix $h = 10$ and vary n. Despite the simplicity of the model, previous work [18] shows that the state-of-the-art model checker Storm [17] is unable to scale beyond $n = 15$. The right diagram in Figure 9 shows how the running time scales with n for Dice and Geni. We set the time limit to 60 minutes for this experiment. The two systems have similar performance up to $n = 15$. Geni performs better beyond that point, whereas Dice does not scale beyond $n = 17$.

The results identify a promising application of Geni: it is potentially beneficial to include Geni (or the techniques embodied by Geni) in the back end of a probabilistic model checking tool.

Grid networks of routers. In this experiment, we evaluate the performance of Geni on a benchmark that Fierens et al. [13] use to demonstrate the scalability of inference in ProbLog, a probabilistic logic programming language.

This benchmark, depicted in Figure 10, models a grid network of $n \times n$ router nodes. Each link between two routers has a probability of failure. The inference problem queries the probability that a packet sent from the top-left router reaches the bottom-right router.

The left diagram in Figure 11 presents the results of this experiment. Genfer and ProbLog do not scale beyond networks containing more than 100 nodes. Dice and Geni scale better. Our

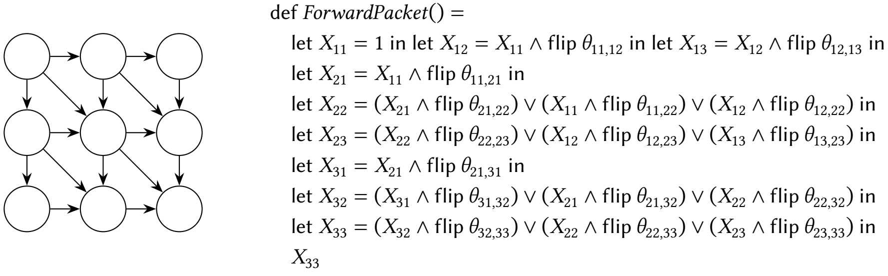


Figure 10. A 3×3 grid-network model with probabilistic link failures. The model is from Fierens et al. [13]. Left: network topology. Right: Geni function modeling the probabilistic behavior of the network.

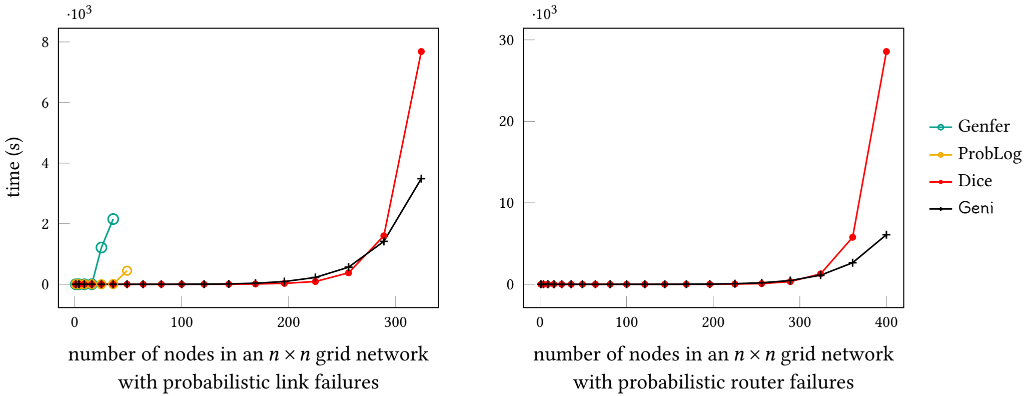


Figure 11. Scaling of Genfer, ProbLog, Dice, and Geni on grid-network benchmarks.

data analysis suggests that Dice's time grows exponentially in n^2, whereas Geni's time grows exponentially in n—Geni exhibits better scalability than Dice on this benchmark.

We also create a benchmark that models an $n \times n$ grid network with probabilistic router failures. The results for Dice and Geni on this benchmark are shown in the right diagram in Figure 11. As in the previous experiment, Geni exhibits better asymptotic performance than Dice.

Bayesian networks in CPT form. Geni is not intended as a direct compilation target for Bayesian networks that are represented as conditional probability tables. Specialized solvers exist for this form of Bayesian networks [31, 11], using *knowledge compilation* techniques [12]. Dice, at its core, follows this approach, compiling weighted Boolean formulas to BDDs, which makes it well suited for handling CPTs.

Although Geni is not designed for CPTs, we still evaluate it on Bayesian network benchmarks [3] where Dice performs well. We directly encode the CPTs as Geni programs, using the same encoding strategy in which they are ported to Dice.

We set a time limit of five minutes for each run. The inference problem is to compute the marginal distribution of a target variable. Since the CPT form of the Bayesian networks does not specify the target variable, for each system, we run inference on all variables in the network and report the maximum time taken.

Table 1 shows the results. Geni and Dice are comparable on models such as Hepar 2 and Pigs, which contain thousands of parameters, but Dice is faster on larger models. Genfer times out on

Table 1. Comparing the performance of three systems on Bayesian networks in CPT form. Time-out (t/o) is set to five minutes (i.e., 300 s).

Benchmarks	Time (s)			Benchmark statistics		
	Genfer	Geni	Dice	# nodes	# arcs	# params
Alarm	t/o	0.026	0.032	37	46	509
Insurance	t/o	1.714	0.631	27	52	1008
Hepar 2	t/o	0.037	0.038	70	123	1453
Hailfinder	t/o	1.472	0.359	56	66	2656
Pigs	t/o	0.034	0.037	441	592	5618
Water	t/o	51.798	0.670	32	66	10083
Munin	t/o	281.203	20.191	1041	1397	80592

all the benchmarks. By contrast, Geni can compute exact solutions within five minutes for all the Bayesian-network models we tested.

Prior work suggests that encoding strategies can strongly influence the performance of Bayesian network inference [7]; encodings may exist that advantageously translate local structure implicit in tabular representations into program structure and improve the performance of Geni on large CPTs.

Poisson packet arrival. Since Geni reuses Genfer's facilities for computing higher-order derivatives, its performance profile on Genfer's benchmarks for infinite-support distributions is similar to Genfer. We do not repeat the experiments here. Instead, we evaluate Geni on a new benchmark that Genfer cannot handle: a packet-arrival model similar to the one introduced at the end of Section 3. This benchmark exercises Geni's support for infinite-support distributions and also the loop–sum construct.

In this model, packets arrive according to a Poisson process. All the packets are forwarded through a network of routers. Here, we use the same 3×3 grid network as in Figure 10. The program is conditioned on observing that all packets are successfully forwarded through the grid network. The inference problem is to compute the posterior expectation of the total number of packets.

Because existing tools for exact inference cannot handle this program, we turn to approximate inference for comparison with Geni. The challenge of this benchmark is further underscored by the limited options of applicable approximate inference methods. Since the model is non-differentiable, methods like Hamiltonian Monte Carlo are not applicable. Sequential Monte Carlo is not applicable either, as the program's control flow is stochastic. This leaves us with broadly applicable but often inefficient methods like importance sampling (IS). Specifically, we use the Pyro PPL [2] and its implementation of IS to estimate the posterior expectation.

Figure 12 shows the results. For Pyro, we plot how the L1 distance between the true posterior expectation and the estimate from IS changes as time progresses. As more samples are collected, the estimate seems to converge to the true posterior expectation, but it only starts to show signs of convergence well after ten minutes. The convergence is slow, because the inference problem has a high-dimensional, discrete latent space. In contrast, Geni swiftly outputs the exact solution.

To our knowledge, Geni is the first system that can automatically derive exact solutions for this class of probabilistic programs. Concurrent work [33] uses a similar model to demonstrate a hybrid inference approach that combines BDD-based exact inference (for the grid network) and importance

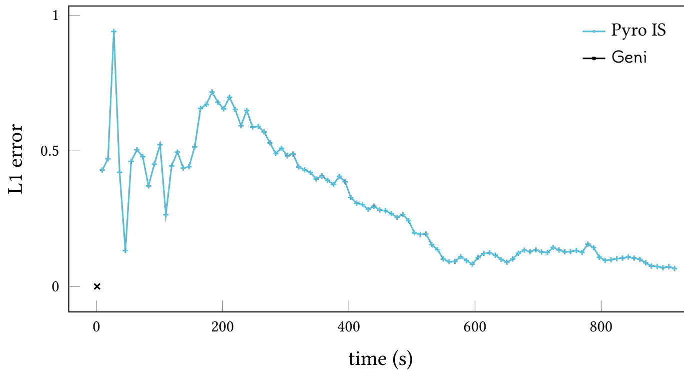


Figure 12. Comparing Pyro's importance sampling and Geni's exact inference on a Poisson packet arrival model. Packets arrive at a rate of 10 and are routed through the grid network shown in Figure 10.

sampling (for the Poisson process and the loop). This approach, for its use of approximate inference, is less efficient and cannot produce exact solutions.

8 Related Work

There is a rich literature on inference methods for probabilistic programming languages. We focus our discussion on exact inference.

Exact inference via generating functions. The use of GFs to scale exact inference is a novel contribution of Geni, as existing GF-based systems focus on goals other than inference scalability. Klinkenberg et al. [21] define a GF semantics for an imperative language and use it to analyze if a given GF overapproximates a program's meaning. Building on this semantics, Chen et al. [8] and Klinkenberg et al. [22] develop techniques for semi-automated analysis of imperative programs with loops. The approach requires the user to provide loop-invariant templates, which is a non-trivial task. This method is implemented in a system called Prodigy and uses a computer algebra system (CAS) in the back end. Genfer, by Zaiser et al. [36], is an imperative language without loops. Inference in Genfer has been shown to be more efficient than in Prodigy, for loop-free programs. The efficiency is attributed to the use of automatic differentiation instead of CAS.

Other approaches to exact inference. Other PPLs use representations and algorithms other than GFs for exact inference. BernoulliProb [10] and the probabilistic model checker Storm [17] use algebraic decision diagrams [1]. Dice [19] uses binary decision diagrams. SPPL [30] uses sum-product networks. FSPN [34] and PERPL [9] handle unbounded recursion by compiling a recursive program to a system of equations and solving it. MAPPL [27] generalizes the variable elimination algorithm [37] to handle evidence-finite recursive programs. Hakaru [29] and Psi [14] use algebraic simplification and symbolic integration, relying on a CAS for finding closed-form solutions.

The design of these PPLs often involves a trade-off between the expressivity of the language and the tractability of inference. Greater expressivity can make exact inference intractable or even impossible—closed-form solutions may not exist in the chosen representation. In the systems mentioned above, those with a more expressive language tend to be less scalable and less predictable in performance, while those with better scalability and predictability tend to support a less expressive language.

9 Conclusion

Exact inference for probabilistic programs is both computationally and analytically challenging. This paper presents a compiler for a functional probabilistic programming language that targets generating functions, improving the scalability of exact inference while enhancing expressive power and ensuring fundamental correctness properties.

We hope this work will inspire further investigation into the use of generating functions for probabilistic inference and verification. Future directions include developing more sophisticated compiler optimizations, further enriching the expressive power of the language, integrating our approach with other exact or approximate inference methods, and applying our techniques to broader domains and inference tasks in science and engineering.

Data-Availability Statement

The software artifact accompanying this paper is available on Zenodo [25] and GitHub [26].

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. We are grateful to Oghenevwogaga Ebresafe for his generous help with the implementation of Geni. We also thank Marco Calabretta and Farzan Bhuiyan for their assistance with the experiments. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada. Any opinions and findings are those of the authors and do not necessarily reflect the position of any funders.

References

[1] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, Gary D. Hachtel, Enrico Macii, Abelardo Pardo, and Fabio Somenzi. 1997. Algebraic decision diagrams and their applications. *Formal Methods in System Design* 10 (1997).

[2] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep universal probabilistic programming. *Journal of Machine Learning Research (JMLR)* 20, 1 (2019). arXiv:1810.09538

[3] BN Repo [n.d.]. Bayesian network repository. https://www.bnlearn.com/bnrepository. Accessed: 2025-02-01.

[4] Randal E. Bryant. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.* 35, 8 (Aug. 1986). https://doi.org/10.1109/TC.1986.1676819

[5] William X. Cao, Poorva Garg, Ryan Tjoa, Steven Holtzen, Todd Millstein, and Guy Van den Broeck. 2023. Scaling integer arithmetic in probabilistic programs. In *Conf.on Uncertainty in Artificial Intelligence (UAI)*.

[6] Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. 2017. Stan: A probabilistic programming language. *Journal of Statistical Software* 76, 1 (2017). https://doi.org/10.18637/jss.v076.i01

[7] Mark Chavira and Adnan Darwiche. 2008. On probabilistic inference by weighted model counting. *Artificial Intelligence* 172, 6 (2008). https://doi.org/10.1016/j.artint.2007.11.002

[8] Mingshuai Chen, Joost-Pieter Katoen, Lutz Klinkenberg, and Tobias Winkler. 2022. Does a program yield the right distribution? Verifying probabilistic programs via generating functions. In *Int'l Conf.on Computer Aided Verification (CAV)*. https://doi.org/10.1007/978-3-031-13185-1_5

[9] David Chiang, Colin McDonald, and Chung-chieh Shan. 2023. Exact recursive probabilistic programming. *Proc.of the ACM on Programming Languages (PACMPL)* 7, OOPSLA1 (April 2023). https://doi.org/10.1145/3586050

[10] Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. 2013. Bayesian inference using data flow analysis. In *Proc.of the 9th Joint Meeting of the European Software Engineering Conf.and the ACM SIGSOFT Symp.on the Foundations of Software Engineering (ESEC/FSE)*. https://doi.org/10.1145/2491411.2491423

[11] Adnan Darwiche. 2009. *Modeling and Reasoning with Bayesian Networks*. Cambridge University Press. https://doi.org/10.1017/CBO9780511811357

[12] Adnan Darwiche and Pierre Marquis. 2002. A knowledge compilation map. *Journal of Artificial Intelligence Research* 17, 1 (Sept. 2002).

[13] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. 2015. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory and Practice of Logic Programming* 15, 3 (2015). https://doi.org/10.1017/S1471068414000076

[14] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact symbolic inference for probabilistic programs. In *Int'l Conf.on Computer Aided Verification (CAV)*. https://doi.org/10.1007/978-3-319-41528-4_4

[15] Michele Giry. 1981. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*. https://doi.org/10.1007/BFb0092872

[16] Maria I. Gorinova, Andrew D. Gordon, Charles Sutton, and Matthijs Vákár. 2021. Conditional independence by typing. *ACM Tran.on Programming Languages and Systems (TOPLAS)* 44, 1 (Dec. 2021). https://doi.org/10.1145/3490421

[17] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. 2022. The probabilistic model checker Storm. *Int'l Journal on Software Tools for Technology Transfer (STTT)* (2022). https://doi.org/10.1007/s10009-021-00633-z

[18] Steven Holtzen, Sebastian Junges, Marcell Vazquez-Chanlatte, Todd Millstein, Sanjit A. Seshia, and Guy Van den Broeck. 2021. Model checking finite-horizon Markov chains with probabilistic inference. In *Int'l Conf.on Computer Aided Verification (CAV)*. https://doi.org/10.1007/978-3-030-81688-9_27

[19] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling exact inference for discrete probabilistic programs. *Proc.of the ACM on Programming Languages (PACMPL)* 4, OOPSLA (Nov. 2020). https://doi.org/10.1145/3428208

[20] Hyperfine [n.d.]. Hyperfine: A command-line benchmarking tool. https://github.com/sharkdp/hyperfine. Accessed: 2025-02-01.

[21] Lutz Klinkenberg, Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Joshua Moerman, and Tobias Winkler. 2020. Generating functions for probabilistic programs. In *Logic-Based Program Synthesis and Transformation (LOPSTR)*. https://doi.org/10.1007/978-3-030-68446-4_12

[22] Lutz Klinkenberg, Christian Blumenthal, Mingshuai Chen, Darion Haase, and Joost-Pieter Katoen. 2024. Exact Bayesian inference for loopy probabilistic programs using generating functions. *Proc.of the ACM on Programming Languages (PACMPL)* 8, OOPSLA1 (April 2024). https://doi.org/10.1145/3649844

[23] Dexter Kozen. 1981. Semantics of probabilistic programs. *J. Comput. System Sci.* 22, 3 (1981). https://doi.org/10.1016/0022-0000(81)90036-2

[24] Marta Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of probabilistic real-time systems. In *Int'l Conf.on Computer Aided Verification (CAV)*.

[25] Jianlin Li, Oghenevwogaga Ebresafe, and Yizhou Zhang. 2025. Compiling with generating functions (artifact). https://doi.org/10.5281/zenodo.16137388

[26] Jianlin Li, Oghenevwogaga Ebresafe, and Yizhou Zhang. 2025. Compiling with generating functions (artifact). https://github.com/geni-icfp25-ae/geni-icfp25-ae

[27] Jianlin Li, Eric Wang, and Yizhou Zhang. 2024. Compiling probabilistic programs for variable elimination with information flow. *Proc.of the ACM on Programming Languages (PACMPL)* 8, PLDI (2024). https://doi.org/10.1145/3656448

[28] Jianlin Li and Yizhou Zhang. 2025. *Compiling with Generating Functions (Extended Version)*. Technical Report CS-2025-03. School of Computer Science, University of Waterloo.

[29] Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. 2016. Probabilistic inference by program transformation in Hakaru (system description). In *Functional and Logic Programming*. https://doi.org/10.1007/978-3-319-29604-3_5

[30] Feras A. Saad, Martin C. Rinard, and Vikash K. Mansinghka. 2021. SPPL: Probabilistic programming with fast exact symbolic inference. In *ACM SIGPLAN Conf.on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/3453483.3454078

[31] Tian Sang, Paul Beame, and Henry Kautz. 2005. Performing Bayesian inference by weighted model counting. In *AAAI Conf.on Artificial Intelligence (AAAI)*. https://doi.org/10.5555/1619332.1619409

[32] Sam Staton. 2020. *Probabilistic Programs as Measures*. Cambridge University Press. https://doi.org/10.1017/9781108770750.003

[33] Sam Stites, John M. Li, and Steven Holtzen. 2025. Multi-language probabilistic programming. *Proc.of the ACM on Programming Languages (PACMPL)* 9, OOPSLA1 (April 2025). https://doi.org/10.1145/3720482

[34] Andreas Stuhlmüller and Noah D. Goodman. 2012. A dynamic programming algorithm for inference in recursive probabilistic programs. (2012). arXiv:1206.3555

[35] Herbert S. Wilf. 2005. *generatingfunctionology* (3 ed.). CRC Press.

[36] Fabian Zaiser, Andrzej S. Murawski, and C.-H. Luke Ong. 2023. Exact Bayesian inference on discrete models via probability generating functions: A probabilistic programming approach. In *Conf.on Neural Information Processing Systems (NeurIPS)*. arXiv:2305.17058

[37] Nevin Lianwen Zhang and David Poole. 1994. A simple approach to Bayesian network computations. In *Proc.of the 10th Canadian Conference on Artificial Intelligence*.

Received 2025-02-27; accepted 2025-06-27